Seminar Report

# Introduction to Performance Engineering in Rust

Lars Quentin

MatrNr: 21774184

Supervisor: Dr. Artur Wachtel

Georg-August-Universität Göttingen
Campus-Institut Data Science / GWDG

October 1, 2023

# Abstract

Due to its high safety focus for a modern systems programming language, Rust is a promising choice for performance-critical applications in the field of High Performance Computing (HPC). This report covers an introduction to the methodology of performance engineering using Rust's still-developing ecosystem. It was done using the concept of problem-based learning, where, using the example of quadratic matrix multiplication, many concepts of performance engineering were explored. This report covers micro benchmarking, full benchmarking, profiling, assembly analysis, compiler optimizations, and an introduction to single-threaded SIMD parallelism as well as multi-threading. While the ecosystem is still experimental and ever-changing, the tooling available is already sufficient for thorough performance analysis.

## Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☑ Not at all
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☐ Other, namely: -

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

# Contents

# List of Tables

# List of Figures

# List of Listings

# List of Abbreviations

| | |
|---|---|
| **ADT** | Algebraic Data Type |
| **CI** | Continuous Integration |
| **HPC** | High-Performance Computing |
| **IR** | Intermediate Representation |
| **I/O** | Input / Output |
| **IoT** | Internet of Things |
| **KDE** | Kernel Density Estimation |
| **LLVM** | Low Level Virtual Machine |
| **LTO** | Link Time Optimization |
| **PBL** | Problem-Based Learning |
| **PDF** | Probability density function |
| **PGO** | Profile Guided Optimization |
| **RAII** | Resource Acquisition Is Initialization |
| **SIMD** | Single Instruction Multiple Data |

# 1 Introduction

## 1.1 Motivation

From a programming language perspective, High-Performance Computing (HPC) is dominated by code written in C, C++, or Fortran as these provide the low-level control and optimization capabilities required for tightly-optimized code. However, as Rust was initially designed as a modern, memory-safe C++ replacement, it could be a valid choice for any kind of performance-critical code.

Rather than just providing yet another taxonomy of successful HPC projects in Rust, this report will give an introduction to the topic of performance engineering in Rust, implicitly covering the current state of the ecosystem while providing a short explanation of each of the common concepts. Instead of just providing an enumeration of techniques, it uses Problem-Based Learning (PBL) to introduce the techniques just in time when they are relevant, providing a more coherent learning progression.

Problem-Based Learning can be difficult. The problem has to be

- Small enough to fit the scope of a report
- Complex enough to cover most of the concepts of real-life performance engineering
- Interesting enough to keep readers engaged in the topic

For this report, we decided to analyze matrix multiplications. More than just a toy problem, matrix multiplication is at the core of all deep learning frameworks. As the parameter count steadily increases into the trillions [1], fast matrix multiplications become evermore important for today's frameworks.

## 1.2 Rust

Rust [2] is a systems programming language initially released by Mozilla Research in 2015. It was designed as a memory-safe alternative for C++ in Servo [3], which is the web rendering engine used in Firefox. Rust's main goal is to provide memory safety while having an on-par performance with other systems languages such as C or C++.

Having memory safety is paramount, as most security issues in traditional C/C++ codebases are a result of using a memory-unsafe language. To quote the overview by Alex Gaynor [4]

- Android [5]: "Our data shows that issues like use-after-free, double-free, and heap buffer overflows generally constitute more than *65%* of High & Critical security bugs in Chrome and Android."
- Android's bluetooth and media components [6]: "Use-after-free (UAF), integer overflows, and out of bounds (OOB) reads/writes comprise *90%* of vulnerabilities with OOB being the most common."

- iOS and macOS [7]: "Across the entirety of iOS 12 Apple has fixed 261 CVEs, 173 of which were memory unsafety. That's *66.3%* of all vulnerabilities." and "Across the entirety of Mojave Apple has fixed 298 CVEs, 213 of which were memory unsafety. That's *71.5%* of all vulnerabilities."
- Chrome [8]: "The Chromium project finds that around *70%* of our serious security bugs are memory safety problems."
- Microsoft [9]: *"∼70%* of the vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues"
- Firefox's CSS subsystem [10]: "If we'd had a time machine and could have written this component in Rust from the start, 51 (*73.9%*) of these bugs would not have been possible."
- Ubuntu's Linux kernel [11]: "*65%* of CVEs behind the last six months of Ubuntu security updates to the Linux kernel have been memory unsafety."

Furthermore, it is now adapted by many big tech firms such as Amazon [12], Google [13], Meta [14], and Microsoft [15]. Lastly, in December 2022, it became the first language besides C and Assembly supported for Linux kernel development [16].

### 1.2.1 Why Rust is a good fit for HPC

One can think of Rust as a modern dialect of C++ enforced by the compiler. It uses Resource Acquisition Is Initialization (RAII) internally to ensure memory safety, while references are roughly equivalent to `std::unique_ptr`.

Especially relevant is the great interoperability with other languages. It supports easy integration with C++ using `bindgen` [17], which is maintained by the Rust core team. Rust also allows for easy embedding into Python code using `PyO3` [18], allowing for highly performant native extensions.

Furthermore, it allows for very low-level control, even to the extent of bare metal deployment support. Due to Rust's aforementioned RAII-alike memory management model, the runtime does not need for a garbage collector. One can even bring their own memory allocator and do raw pointer arithmetic if required. Lastly, Rust supports architecture-based conditional compilation which makes it possible to write fast programs leveraging modern CPU instructions while providing portable alternatives. To support bare metal, OS-less development, Rust's standard library is split into 3 tiers:

- `core`: The `core` library provides essential types and functionality that do not require heap memory allocation.
- `alloc`: The `alloc` library builds upon the `core` library but expects heap allocations, thus supporting things such as dynamically sized vectors.
- `std`: The `std` library is the highest-level tier, requiring not only a memory allocator but also several OS capabilities such as I/O management.

Although Rust itself is a relatively new language, its compiler supports most modern compiler optimizations. This is possible through Low Level Virtual Machine (LLVM). Instead of producing native assembly for all architectures, the compiler just provides a LLVM frontend generating LLVM Intermediate Representation (IR) which then gets translated to native code by LLVM.

Lastly, it supports many modern functional concepts such as immutability by default, flat traits instead of deep inheritance, exhaustive pattern matching with Algebraic Data Types (ADTs) sum types as well as providing alternatives to nullability, which is commonly known as the billion-dollar mistake [19]. Its language design is in fact so popular that according to the yearly StackOverflow surveys it was voted as the most loved language for the 7th year in a row [20].

### 1.3 Quadratic Matrix Multiplication

Let $A, B \in \mathbb{R}^{n \times n}, n \in \mathbb{N}$. Then the matrix multiplication $C \in \mathbb{R}^{n \times n}$ is defined as

$$C_{ij} := \sum_{k=1}^{n} A_{ik} \cdot B_{kj}.$$

One can think of $C_{ij}$ as the dot product of the $i$-th row of $A$ and the $j$-th column of $B$.

### 1.4 Structure

This report is structured as follows: Section 2 will explore a simplified version of the matrix multiplication problem where the dimension is fixed. Here, the focus will be set on micro benchmarking, full application benchmarking, and assembly analysis. Section 3 will then explore the full matrix multiplication, exploring the topics of profiling, compiler optimizations, cache-oblivious algorithms as well as how to benchmark in noisy environments. Section 4 will provide a short introduction to parallelism. Lastly, section 5 concludes this report by providing an overview of all shown tools as well as further resources.

# 2 Fixed Size Matrix Multiplication

To start with a simplified problem, this section focuses on a fixed quadratic matrix size of $n = 3$. Using the mathematical definition, this can be trivially implemented:

```rust
fn matmul(a: Vec<Vec<f32>>, b: Vec<Vec<f32>>) -> Vec<Vec<f32>> {
    let mut result = vec![vec![0.0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            for k in 0..3 {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    result
}
fn driver_code(a: Vec<Vec<f32>>, b: Vec<Vec<f32>>, c: Vec<Vec<f32>>)
    -> Vec<Vec<f32>> {
    matmul(matmul(a, b), c) // D := A * B * C
}
```

Listing 1: Naive implementation of a $3 \times 3$ matrix multiplication.

The `Vec` arguments are currently passed as call-by-value, which means that the vector struct gets copied onto the function's stack. Intuitively, this could be improved by using call-by-reference semantics, which just copies the pointer instead of the underlying data. Theoretically, this should result in a performance improvement. In reality, it is very hard to predict actual performance. Thus, some benchmarking is required. To measure the performance, either micro benchmarking or full application benchmarking can be used.

## 2.1 Micro benchmarking

Micro benchmarking is the performance evaluation of small isolated functions. In the Rust ecosystem, there are two obvious solutions for micro benchmarking: Rust's native `cargo bench` as well as `criterion.rs`, which is the modern canonical benchmark library.

**Native Benchmarking** Cargo, Rust's package manager, supports benchmarking natively through the `cargo bench` [21] subcommand. Unfortunately, this is still experimental, thus only part of the unstable nightly Rust versions. Furthermore, no clear roadmap to stability exists [22].

`cargo bench` is a very lightweight microbenchmarking solution. It provides no integrated regression testing or any kind of visualization or plotting. The 3rd-party `cargo-benchcmp` [23] utility can be used to compare different benchmarks.

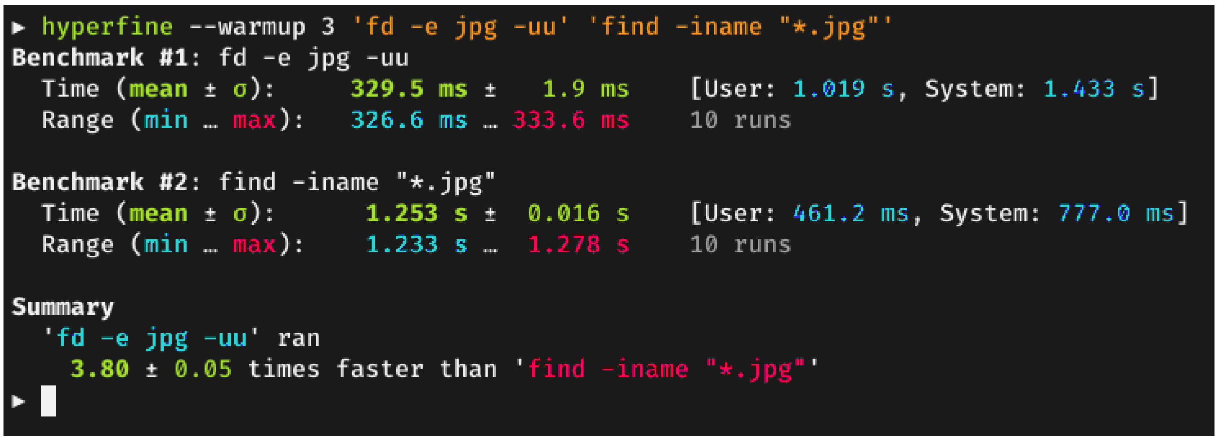
**Criterion** The other solution is `criterion.rs` [24], which is also available in stable Rust. It uses basic statistical outlier detection to measure regressions and their significance. Furthermore, it blocks constant folding using the `criterion::black_box`, which is described as a “function that is opaque to the optimizer, used to prevent the compiler from optimizing away computations in a benchmark” [25]. It automatically generates HTML reports with plots using `gnuplot` or `plotters`. For benchmark comparisons, the `cargo-critcmp` [26] program can be used.

As there is currently no active development in `cargo bench`, criterion should always be the preferred solution for micro benchmarking.

## 2.2 Full Application Benchmarking

There are several solutions for benchmarking whole applications, especially as they are usually agnostic to the application's programming language. But to stick to the modern Rust ecosystem, this report will focus on Hyperfine [27], a very actively developed command-line benchmarking tool written in Rust.

From a simplified perspective, full application benchmarking is quite trivial. First, take a timestamp of the current time. Then, run the command to be benchmarked. Afterwards, take a new timestamp. The time delta is the benchmark time. But beyond this core functionality, Hyperfine supports many important and fundamental features for proper benchmarking and analysis.



```
▶ hyperfine --warmup 3 'fd -e jpg -uu' 'find -iname "*.jpg"'
Benchmark #1: fd -e jpg -uu
  Time (mean ± σ):     329.5 ms ±   1.9 ms    [User: 1.019 s, System: 1.433 s]
  Range (min … max):   326.6 ms … 333.6 ms    10 runs

Benchmark #2: find -iname "*.jpg"
  Time (mean ± σ):      1.253 s ±  0.016 s    [User: 461.2 ms, System: 777.0 ms]
  Range (min … max):    1.233 s …  1.278 s    10 runs

Summary
  'fd -e jpg -uu' ran
    3.80 ± 0.05 times faster than 'find -iname "*.jpg"'
▶ 
```

Figure 1: An example picture of Hyperfine's output comparing `fd` and `find` [27]

Firstly, Hyperfine supports out-of-the-box statistical analysis and outlier detection. Since it can be assumed that the program run times are approximately equal, benchmark times should be normally distributed. Thus, by fitting a normal distribution over all runs and computing its confidence interval, it can detect any outliers. Secondly, it allows for warmup runs and cache-clearing commands [1] between each run. Warmup runs are useful to fill caches such as the page cache for disk I/O. Lastly, it supports further analysis by providing an export to various formats, such as CSV, JSON, Markdown, or AsciiDoc, which can then be analyzed programmatically. Hyperfine's repository contains several Python scripts for basic visualization [29], which can be used as a starting point for further analysis.

## 2.3 Performance Optimization

For the benchmarking, the aforementioned `criterion.rs` benchmarking framework is used. The benchmark was done on a Dell Latitude 7420 with an Intel i5-1145G7 and 16GB of LPDDR4 RAM, compiled with rustc 1.72.0. The CPU idle was around 1%.

[1] such as `echo 1 > /proc/sys/vm/drop_caches` to free the page cache [28].

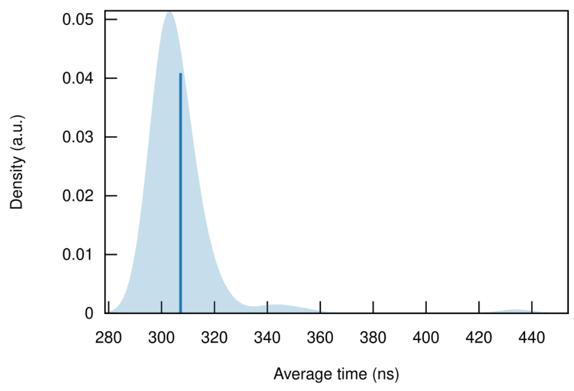
### 2.3.1 Call By Reference

Although the data part of `Vec<>` is stored on the heap[2] the stack part is still very complex since it has to keep track of several things such as the current size and its current maximal capacity. More importantly, the `Vec<>` struct has a bigger memory footprint than a pointer. Thus, using Call-By-Reference should improve the performance by requiring fewer memory copies! In Rust, this can be archived using the `&` operator:

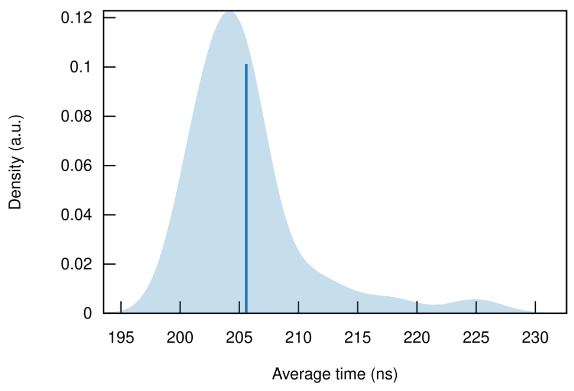
```rust
fn matmul(a: &Vec<Vec<f32>>, b: &Vec<Vec<f32>>) -> Vec<Vec<f32>> {
    /* Only the signature changes... */
}
fn driver_code(a: Vec<Vec<f32>>, b: Vec<Vec<f32>>, c: Vec<Vec<f32>>)
    -> Vec<Vec<f32>> {
    matmul(matmul(a, b), c) // D := A * B * C
}
```

Listing 2: Changing the signature to Call-By-Reference semantics with references.

Using the default criterion settings [3], the following results were benchmarked:



(a) Call By Value: Mean 307.19ns



(b) Call By Reference: Mean 205.58ns

Figure 2: Comparsions of the PDFs computed using KDE for Call-By-Value and Call-By-Reference.

According to the benchmarks, this change alone resulted in a 49.426% mean increase! Note that the full metric table is included in the appendix.

There is one more obvious possible improvement to try: In this code, Rust's `Vec<>`, a dynamically sized, heap-allocated vector is used. This could be replaced with a normal C-type array.

[2]Since `Vec<>` are dynamically resizable, the absolute size can't be known at compile time.
[3]Default Rust Release build and 100 iterations.

### 2.3.2 Primitive Stack Arrays

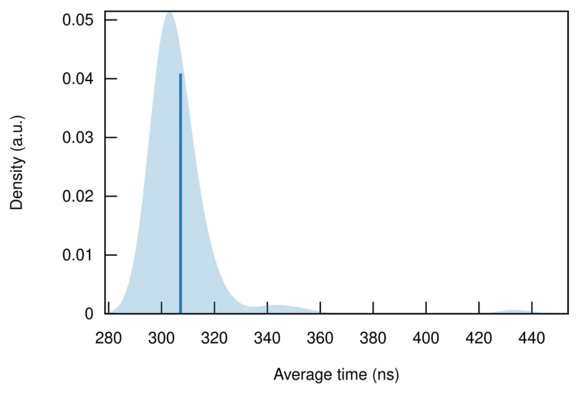
Compared to static arrays, `Vec<>` has much more overhead. Firstly, since its size is not known at compile time, it performs several run-time bounds checks [4]. Next, it has to be heap-allocated, which can be way more expensive and results in worse memory locality. Lastly, `Vec<>` is a complex struct with many functions and features, which in turn results in more computation required.

This is the code using primitive stack arrays instead of the more sophisticated, dynamically allocated vectors:

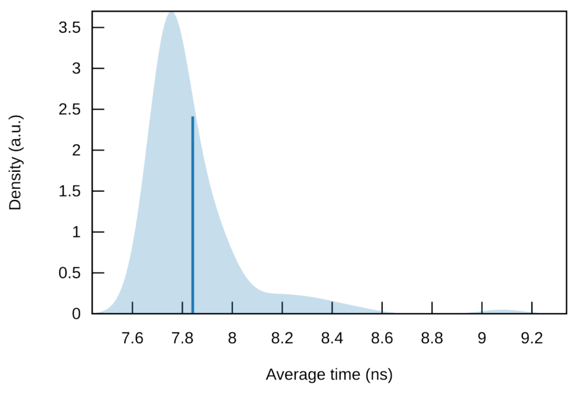
```rust
fn matmul(a: &[[f32; 3]; 3], b: &[[f32; 3]; 3], result: &mut [ [f32; 3]; 3]) {
    for i in 0..3 {
        for j in 0..3 {
            for k in 0..3 {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}

fn driver_code(a: &[[f32; 3]; 3], b: &[[f32; 3]; 3], c: &[[f32; 3]; 3],
               res_buf: &mut [[f32; 3]; 3]) {
    let mut temp = [[0.0; 3]; 3];
    matmul3(a, b, &mut temp);
    matmul3(&temp, c, res_buf);
}
```

Listing 3: Changing the signature to Call-By-Reference semantics with references.

Here are the results, compared to the initial version:



(a) Call By Value: Mean 307.19ns



(b) Primitive Arrays: Mean 7.841ns

Figure 3: Comparsions of the PDFs computed using KDE for the initial version and the one using primitive arrays.

[4]This can be partially avoided. For more information, see the Bounds Checks Cookbook [30]

This results in a staggering 3831.27% mean increase! Once again, note that the full metric table is included in the appendix.

There are several possible explanations for those results. It could be that the bounds check prevents instruction pipelining. But the main reason is most likely that the prior version requires an expensive heap allocation for the return value while the static arrays are already preallocated on the stack [5]. Now that all high-level optimizations are applied, the next step would be to optimize on the assembly-level. The next section will show how to use proper tooling for assembly level optimization in Rust.

## 2.4 Assembly Optimizations

Modern compilers, such as the LLVM based rustc, do a lot of optimization for performance. This results in vastly different assemblies when comparing unoptimized code (`-O0`) to their optimized counterpart (`-O3`). Thus, it often makes sense to look at the assembly for hot code paths[6]. Naively, one could just look at the whole binary and decode the bytes into their instructions. This is neither useful nor reasonable for large programs to analyze. Instead, in this section, two different ways to analyze assembly will be analyzed: The well-known Compiler Explorer [31] as well as the `cargo-show-asm` crate [32].

**Compiler Explorer** Compiler Explorer [31] is an online development environment initially developed by Matt Godbolt, and primarily used for analysis of C and C++ applications. It was started in 2023 to optimize financial quantitative analysis algorithms. Compiler Explorer supports over 30 different languages; from typical high-performance languages such as C, C++, and Fortran, bytecode languages such as Python and Java to more niche languages such as Haskell and Solidity. Additionally, one can compare different compilers (for example `gcc`, `clang`, and `msvc` for C applications) and manually specify different compiler arguments such as `-Osize` instead of `-O3`. Since it can also be hosted on-premise, proprietary and other custom compilers can be added if needed.

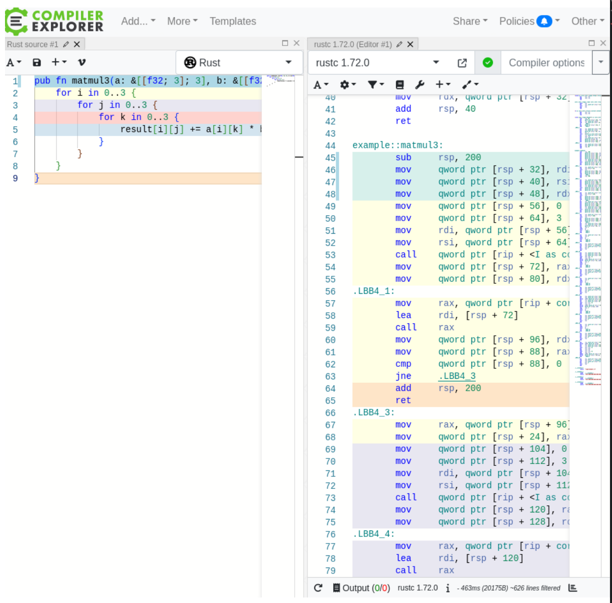
Its main feature is the color coding; Compiler Explorer assigns each function line to a specific color. The same color will then be used in the assembly window, providing an intuitive mapping and a good overall user experience. Unfortunately, it does not support multiple files and hasn't any dependency management [7]. To summarize, Compiler Explorer is the best fit for small, single-file programs. For larger applications, the next tool can be used.

**cargo-show-asm** `cargo-show-asm` [32] is a more minimalist, less polished console application for analyzing assembly. It works with any rust code base, no matter the size or amount of dependencies. Instead of showing the assembler for all functions, one can query single functions as a CLI parameter. Lastly, instead of the architecture-specific assembler code, it can also return the LLVM IR instead.

[5]Further analysis could be done through statistical profiling, which will be explained later. However since it doesn't help explain performance engineering concepts, it is left as an exercise to the reader.

[6]Code paths are 'hot' when they are executed very frequently, therefore crucial for the overall performance.

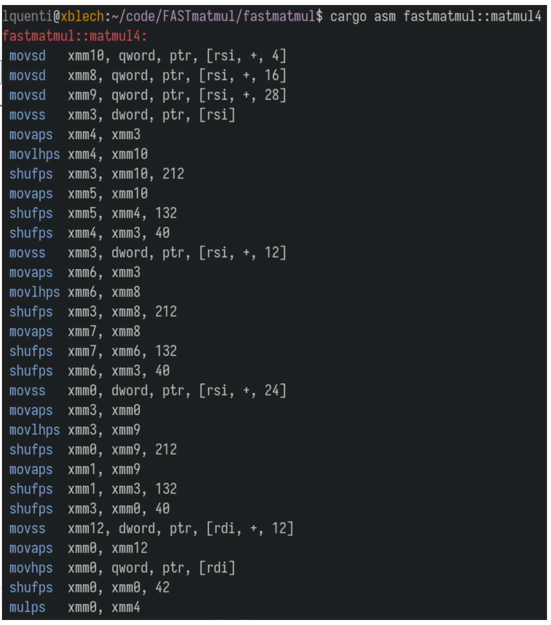
[7]However one could work around this restriction by installing all dependencies globally on a self-hosted instance.



```
pub fn matmul3(a: &[[f32; 3]; 3], b: &[[f3
    for i in 0..3 {
        for j in 0..3 {
            for k in 0..3 {
                result[i][j] += a[i][k] * b
            }
        }
    }
}
```

```
        mov     rdx, qword ptr [rsp + 32]
        add     rsp, 40
        ret

example::matmul3:
        sub     rsp, 200
        mov     qword ptr [rsp + 32], rdi
        mov     qword ptr [rsp + 40], rsi
        mov     qword ptr [rsp + 48], rdx
        mov     qword ptr [rsp + 56], 0
        mov     qword ptr [rsp + 64], 3
        mov     rdi, qword ptr [rsp + 56]
        mov     rsi, qword ptr [rsp + 64]
        call    qword ptr [rip + <I as co
        mov     qword ptr [rsp + 72], rax
        mov     qword ptr [rsp + 80], rdx
.LBB4_1:
        mov     rax, qword ptr [rip + co
        lea     rdi, [rsp + 72]
        call    rax
        mov     qword ptr [rsp + 96], rdx
        mov     qword ptr [rsp + 88], rax
        cmp     qword ptr [rsp + 88], 0
        jne     .LBB4_3
        add     rsp, 200
        ret
.LBB4_3:
        mov     rax, qword ptr [rsp + 96]
        mov     qword ptr [rsp + 24], rax
        mov     qword ptr [rsp + 104], 0
        mov     qword ptr [rsp + 112], 3
        mov     rdi, qword ptr [rsp + 104
        mov     rsi, qword ptr [rsp + 112
        call    qword ptr [rip + <I as co
        mov     qword ptr [rsp + 120], ra
        mov     qword ptr [rsp + 128], rd
.LBB4_4:
        mov     rax, qword ptr [rip + co
        lea     rdi, [rsp + 120]
        call    rax
```

(a) Compiler Explorer



```
lquenti@xblech:~/code/FASTmatmul/fastmatmul$ cargo asm fastmatmul::matmul4
fastmatmul::matmul4:
 movsd   xmm10, qword, ptr, [rsi, +, 4]
 movsd   xmm8, qword, ptr, [rsi, +, 16]
 movsd   xmm9, qword, ptr, [rsi, +, 28]
 movss   xmm3, dword, ptr, [rsi]
 movaps  xmm4, xmm3
 movlhps xmm4, xmm10
 shufps  xmm3, xmm10, 212
 movaps  xmm5, xmm10
 shufps  xmm5, xmm4, 132
 shufps  xmm4, xmm3, 40
 movss   xmm3, dword, ptr, [rsi, +, 12]
 movaps  xmm6, xmm3
 movlhps xmm6, xmm8
 shufps  xmm3, xmm8, 212
 movaps  xmm7, xmm8
 shufps  xmm7, xmm6, 132
 shufps  xmm6, xmm3, 40
 movss   xmm0, dword, ptr, [rsi, +, 24]
 movaps  xmm3, xmm0
 movlhps xmm3, xmm9
 shufps  xmm0, xmm9, 212
 movaps  xmm1, xmm9
 shufps  xmm1, xmm3, 132
 shufps  xmm3, xmm0, 40
 movss   xmm12, dword, ptr, [rdi, +, 12]
 movaps  xmm0, xmm12
 movhps  xmm0, qword, ptr, [rdi]
 shufps  xmm0, xmm0, 42
 mulps   xmm0, xmm4
```

(b) `cargo-show-asm`

Figure 4: Rust assembly analysis: Compiler explorer and `cargo-show-asm`.

Next, two common compiler optimizations will be covered: Loop Unrolling and Function inlining.

### 2.4.1 Loop Unrolling

On the surface, loop unrolling is a simple concept: If the number of loop iterations is known at compile time, replicate the inner code that amount of time. This has multiple benefits. Firstly, it reduces the number of comparisons and jumps made in every iteration. Secondly, it allows for easier pipelining since there is no need for path prediction anymore! The main drawback is that code duplication increases binary size.

Looking at the assembly, loop unrolling was already applied in our case. But if the compiler did not unroll the function, there are two main ways to force it manually:

- Unroll [33] is a Rust macro to unroll the applications at compile time by replacing the unrolled Rust code in preprocessing. Currently, it can just detect loops with integer literal bounds.
- For more sophisticated loops, LLVM can be configured to more aggressively apply loop unrolling. This is controlled by the `-unroll-threshold` parameter[8].

As an important remark, do not manually apply loop unrolling without benchmarking, as it can worsen performance by consuming too much of the CPU instruction cache.

### 2.4.2 Function Inlining

The next, very common compiler optimization is function inlining. Instead of jumping into a subroutine, executing it, and then returning to the previous instructions it places the assembly of the subroutine into the outer function. On the upside, this eliminates

[8]So to apply it with `rustc`, use `-C llvm-args=-unroll-threshold=N` where `N` is an integer.

the calling overhead (such as moving arguments into registers) specified by the calling convention [9]. As already mentioned above, function inlining can also worsen performance through increased binary size and consequently less efficient cache utilization. In our case, it was not applied[10].

After covering benchmarking and assembly analysis, now the harder problem of variadic size matrix multiplication can be approached!

# 3 Variadic Size Matrix Multiplication

Now, after having a simplified toy problem, let's say the following task is given:

"Our department has built a deep learning framework in Rust that inferences too slowly. Please try to optimize the overall performance of this project"

Before being able to optimize the code, one has to first ask the following question: Why is it so slow? To answer this question, profiling is needed.

## 3.1 Profiling

Profiling is used to find out which parts of the program are executed frequently enough to affect runtime performance[11]. Since Rust produces normal binaries, most traditional profilers just work, including common ones such as Linux `perf` [35] and `cachegrind` [36]. See the profiling section in the Rust Performance Book for a more exhaustive list [37].

Since rust supports polymorphism[12] name mangling occurs. As assembly labels have to be unique, name mangling is a technique used to map all polymorphic instances of a function to a unique assembler label. If the profiler doesn't support unmangling natively, `rustfilt` can be used manually [38].

In this chapter, `cargo-flamegraph` [39] and later `iai` [40] will be used for profiling.

## 3.2 Cargo Flamegraph

Cargo flamegraph [39] is a statistical profiler that creates a flamegraph to analyze. In order to understand the result, one needs to understand how statistical profilers work internally. A statistical profiler works by interrupting the program randomly using the kernels interrupt system[13]. After interrupting, it looks at the stack frame, finding out which function stack is currently called. This is a single data point. Now, using a Monte Carlo approach, a statistical profiler can approximate how much time is spent on each

[9]Interestingly, Rust does not have a default non-FFI calling convention for performance benefits. Instead, it requires all dependencies to be compiled with the same version, which is the canonical way when the package manager `cargo` is used.

[10]But it can be applied manually with the `#[inline(/always/never)]` attribute [34].

[11]So-called **hot paths**.

[12]Both ad-hoc polymorphism through traits and parametric polymorphism based on generics.

[13]Cargo flamegraph uses `perf` and `dtrace` internally.

function by taking many measurements.

A flamegraph is a visualization of the stack frames. The wider the current stack frame, the more often those functions were called when interrupted. One layer got called by the layer below. Here is an example of a flamegraph:

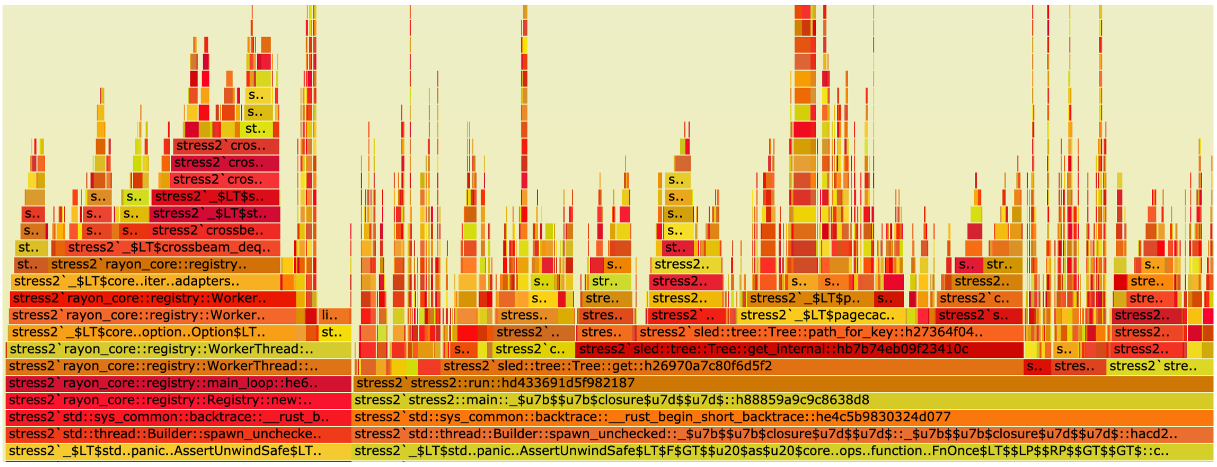


Figure 5: An example flamegraph generated from Rust code [39].

For our fictional deep learning framework, let's assume that the result was a slow $N \times N$ variadic size matrix multiplication! For the benchmarks, $N = 1024$ is used.

## 3.3 Applying Previous Knowledge

Let's say that the matrix multiplication function looks as follows:

```rust
fn matmul(a: Vec<Vec<f32>>, b: Vec<Vec<f32>>) -> Vec<Vec<f32>> {
    let n = a.len();
    let mut result = vec![vec![0.0; n]; n];
    for i in 0..n {
        for j in 0..n {
            for k in 0..n {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    result
}
```

Listing 4: The unoptimized Rust code providing the variadic size quadratic matrix multiplication

Applying the knowledge of our previous chapters, it can already be refactored into the following:

```rust
fn matmul(a: &[f32], b: &[f32], result: &mut [f32], n: usize) {
    for i in 0..n {
        for j in 0..n {
            for k in 0..n {
                result[i * n + j] += a[i * n + k] * b[k * n + j];
            }
        }
    }
}
```

Listing 5: The code optimized analogously to chapter 2.

But before benchmarking this code, there are further free improvements to be had by configuring the compiler to maximize performance.

## 3.4 Compiler Optimizations

Several performance optimizations should be enabled if performance is a high priority:

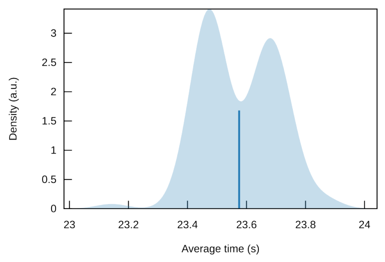
- **Release Builds**: This is by far the biggest improvement. If one does not use the release build[14] the code is not optimized. This enables several general optimizations as well as automatic vectorization.
- **LLVM Link Time Optimization (LTO)**: LTO enabled further, intermodular optimizations during the link stage. While this could improve code by optimizing beyond library bounds, it increases compile time, which is why it is disabled by default.
- **Compiling for Native Architecture:** When compiling for the native architecture[15] the compiler can use more specialized instructions that are not available on every processor such as bigger vector registers for SIMD. Note that this makes the code incompatible with older processor generations.
- **Using a single LLVM codegen unit:** Codegen units are analogous to translation units. This means that, when changing a single file, just the codegen unit in that file has to be recompiled. Therefore, optimizations can't be done beyond codegen unit bounds! Using a single codegen for the whole project allows the compiler to more aggressively optimize globally. Note that this effectively disables partial compilations.
- **Profile Guided Optimization (PGO):** PGO could furthermore be used for more effective branch prediction[16].

[14]With `cargo build -release`.

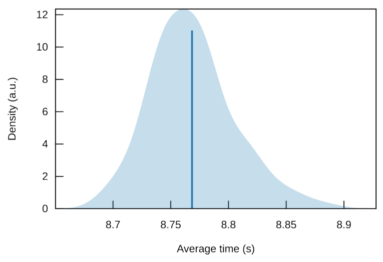
[15]Using the `RUSTFLAGS` environment variable, i.e. `RUSTFLAGS="-C target-cpu=native" cargo build -release`.

[16]This is out of the scope of this project, for an introduction see how the compiler team used it on `rustc` [41]

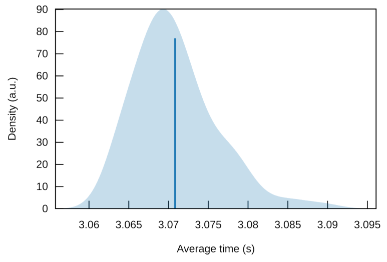
Here are the results, the full table can be found in the appendix:



(a) No compiler optimizations. Naive code. Mean: 23.575s



(b) No compiler optimizations. Optimized code. Mean: 8.7684s



(c) Compiler optimizations. Optimized code. Mean: 3.0708s

Figure 6: Performance comparison between unoptimized code, optimized code and optimized code with compiler optimization enabled.

For the next and last optimization, some further theory is needed.

## 3.5 Cache-oblivious Algorithms

When doing a standard matrix multiplication $C := A \cdot B$, $A$ traverses the matrix in row-major order and $B$ traverses the matrix in column-major order. Since the memory is aligned in row-major order, in every step of $B$ cache miss happens for large sizes of $B$.

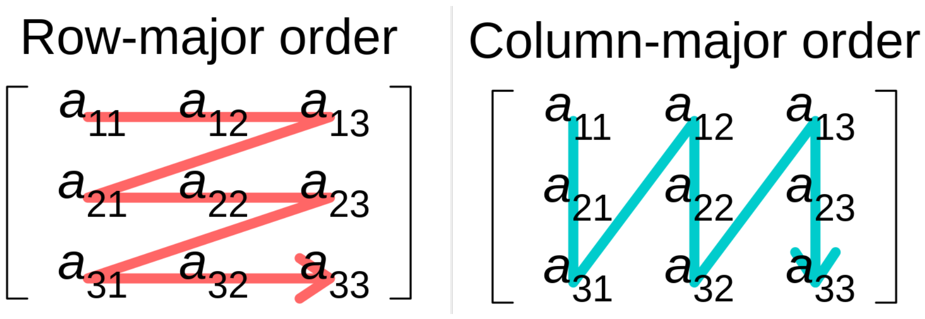


Figure 7: A visualization of row- and column-major order.

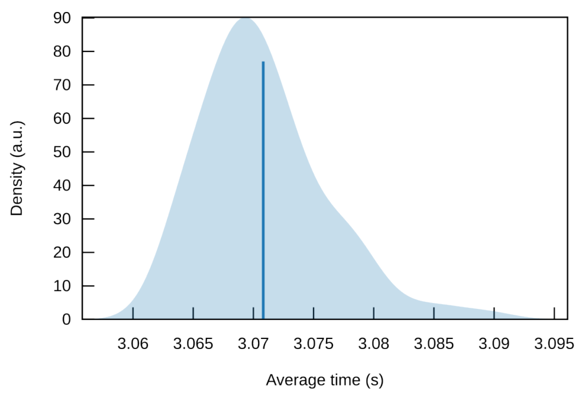
A possible solution could be to compute $A \cdot B^T$ instead by transposing $B$. Then, $C_{ij}$ is computed as row $A_i$ times **row** $B_j$! Unfortunately, since the transpose has to actually be computed beforehand, this requires $\Theta(n^2)$ precompute.

Naturally, two questions arise:

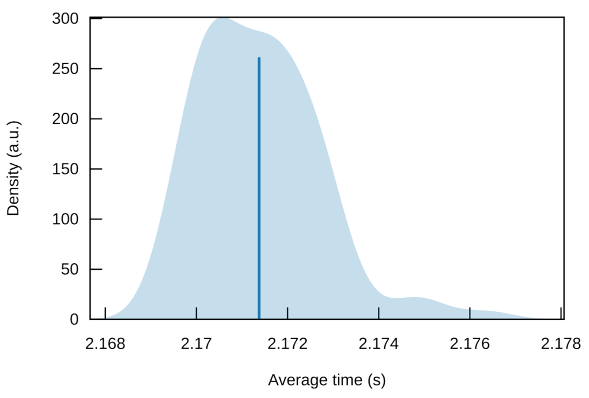
1. Does it improve speed?

2. Does it actually reduce cache misses?

Whether it improves speed can easily be benchmarked.



(a) Compiler optimizations.
Optimized code.
Mean: 3.0708s



(b) Compiler optimizations.
Optimized and transposed code.
Mean: 2.1714s

Figure 8: Performance comparison between unoptimized code, optimized code and optimized code with compiler optimization enabled.

To find out whether it reduces cache misses, `iai` can be used.

## 3.6 Iai

`Iai` [40] is a high-precision, one-shot benchmark framework for Rust code. One-shot means that the code is only run a single time. This is possible by leveraging cachegrind [36] under the hood to simulate the CPU and its caches, allowing us to count all cache accesses. Furthermore, it can be used for Continuous Integration (CI) pipelines since it solves the noisy-neighbour problem of multiple jobs executed on the same runner. Here are the results:

```
iai_normal
  Instructions:         13970975862
  L1 Accesses:          17192372607
  L2 Accesses:           1074884737
  RAM Accesses:              262191
  Estimated Cycles:     22575972977

iai_transpose
  Instructions:          9144912377
  L1 Accesses:          12838193034
  L2 Accesses:             68158189
  RAM Accesses:              328137
  Estimated Cycles:     13190468774
```

Listing 6: The results running iai

Here one can see that, although it uses more RAM accesses, it has better L1 and L2 utilization.

# 4 A glimpse of (Intra-Node) Parallelism

Rust has many ways to do intra-node parallelism[17]. One can categorize intra-node parallelism into two categories: Single-thread parallelism using SIMD instructions and multi-threading. In this chapter, a high-level overview of how to archive both will be given.

## 4.1 SIMD

While a complete introduction to Single Instruction Multiple Data (SIMD) programming would be a report on its own[18] it is noteworthy to say that Rust has two different approaches to support SIMD programming. The old, processor-specific API and the new, portable SIMD API.

**Processor-specific API:** The processor-specific API is experimental only and classified as unsafe, i.e. it doesn't give any memory guarantees. It is composed of the direct low-level intrinsics provided by the CPU manufacturer and has the same function definitions as the C API. The only reason to use this API is that it is part of the `core` instead of the `std` library: This means, it does not expect any memory allocator nor any OS syscalls to work properly and can thus be used for bare metal development.

**Portable SIMD:** The portable SIMD API is, while also experimental, memory-safe. Instead of providing instructions for any CPU architecture, it is generalized on a bit level with types such as `std::simd::{f32x8, f64x4, i32x8}`. This makes it the preferred API for non-bare metal programming. It is part of the `std` library.

Lastly, note that code that does not have those processor features will produce undefined behaviour. There are two ways to mitigate this: If the target architecture is known at compile time, Rust supports conditional compilation[19][20] to provide alternatives to architecture-specific code. If this is not the case, functions such as `std::is_x86_feature_detected` can be used. Note that these should not be used in hot loops as they provide a runtime overhead.

## 4.2 Multithreading

Rust supports several ways of doing multi-threading. First of all, the standard library offers many primitives around simple OS threads[21] See the "Fearless Concurrency" chapter of the Rust book for an introduction [45].

[17]Rust also supports inter-node parallelism using `rsmpi` [42], a rust-native MPI library compatible with OpenMPI and MPICH. Unfortunately, this is out of scope here. For more information, see my report on walky [43], the rusty TSP solver.

[18]For a great introduction on how to do SIMD in Rust, see the SIMD Rust on Android Talk by Guillaume Endignoux [44].

[19]Compile for specific architecture: `#[cfg(target_arch="x86_64")]`

[20]Compile for specific feature: `#[cfg(target_feature="aes")]`

[21]Rust also supported green threads before 1.0 but they were cut because the scheduling meant that they were not zero cost.

Furthermore, Rust provides an `async/await` pattern for managing async Input / Output (I/O). To enable the usage in many, vastly different environments such as Internet of Things (IoT), Rust requires the developer to bring their own async runtime. Most projects use `Tokio` [46], although other projects such as the simpler `smol` or `fuchsia-async` [47] used in Google Fuchsia. For more information on `async/await`, see the official async book [48], the announcement talk from WithoutBoats [49], or fasterthanlime's "Understanding Rust futures by going way too deep" [50].

Lastly, there are several utility libraries. Here, the focus will be on `rayon` because of its simplicity.

### 4.2.1 Rayon

Rayon is a high-level parallelism library using dynamically sized thread pools. It guarantees **data-race freedom** by allowing only one thread to write at a time. Its main features are drop-in parallel iterators: By replacing `.iter()` with `.par_iter`, it is possible to use all functions provided for iterators, such as `.map()`, `.filter()`, `.reduce()` for typical functional patterns or `.join(|| a(), || b()` enabling the `fork-join` computation model. It is best explained with a code example. To rewrite our function in a more iterator-based version:

```rust
fn matmul3(a: &[f32], b: &[f32], result: &mut [f32], n: usize) {
    result.iter_mut().enumerate().for_each(|(idx, res)| {
        let i = idx / n;
        let j = idx % n;
        *res = (0..n).map(|k| a[i * n + k] * b[k * n + j]).sum();
    });
}
```

Listing 7: An more functional version of our matrix multiplication

This can be parallelized by only replacing `.iter_mut()` with `.par_iter_mut()`:

```rust
fn matmul3(a: &[f32], b: &[f32], result: &mut [f32], n: usize) {
    result.par_iter_mut().enumerate().for_each(|(idx, res)| {
        let i = idx / n;
        let j = idx % n;
        *res = (0..n).map(|k| a[i * n + k] * b[k * n + j]).sum();
    });
}
```

Listing 8: The parallelized version, changing a single function call.

## 5 Conclusion and Further Ressources

To conclude, although many parts are still experimental, all tooling for proper performance engineering exists, making Rust a viable programming language for HPC. A summary of all tools can be found in the appendix.

If one is more interested in performance engineering, the best resource for understanding performance tuning in Rust is the "Rust Performance Book" [51]. To understand more about performance engineering and the theory behind it, the free online book "Algorithmica: Algorithms for Modern Hardware" [52] is a good starting point. Finally, if one is more interested in inter-node parallelism, the `rsmpi` library [42] has great examples to get started.

The full code can be found at `https://github.com/lquenti/IntroPerfEng`.

# References

[1] Vitalii Shevchuk. *GPT-4 Parameters Explained: Everything You Need to Know.* Medium. July 17, 2023. URL: `https://levelup.gitconnected.com/gpt-4-parameters-explained-everything-you-need-to-know-e210c20576ca` (visited on 08/15/2023).

[2] *Rust Programming Language.* URL: `https://www.rust-lang.org/` (visited on 08/15/2023).

[3] The Servo Project Developers. *Servo, the parallel browser engine.* Servo. URL: `https://servo.org/` (visited on 08/15/2023).

[4] Alex Gaynor. *What science can tell us about C and C++'s security · Alex Gaynor.* URL: `https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/` (visited on 08/15/2023).

[5] *Detecting Memory Corruption Bugs With HWASan.* Android Developers Blog. URL: `https://android-developers.googleblog.com/2020/02/detecting-memory-corruption-bugs-with-hwasan.html` (visited on 08/15/2023).

[6] *Queue the Hardening Enhancements.* Google Online Security Blog. URL: `https://security.googleblog.com/2019/05/queue-hardening-enhancements.html` (visited on 08/15/2023).

[7] *Memory Unsafety in Apple's Operating Systems.* URL: `https://langui.sh/2019/07/23/apple-memory-safety/` (visited on 08/15/2023).

[8] *Memory safety.* URL: `https://www.chromium.org/Home/chromium-security/memory-safety/` (visited on 08/15/2023).

[9] *A proactive approach to more secure code | MSRC Blog | Microsoft Security Response Center.* URL: `https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/` (visited on 08/15/2023).

[10] *Implications of Rewriting a Browser Component in Rust – Mozilla Hacks - the Web developer blog.* Mozilla Hacks – the Web developer blog. URL: `https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust` (visited on 08/15/2023).

[11] Geoffrey Thomas [@geofft]. *Some unofficial @LazyFishBarrel stats from @alex_gaynor and myself: 65% of CVEs behind the last six months of Ubuntu security updates to the Linux kernel have been memory unsafety.* Twitter. May 26, 2019. URL: `https://twitter.com/geofft/status/1132739184060489729` (visited on 08/15/2023).

[12] *Why AWS loves Rust, and how we'd like to help | AWS Open Source Blog.* Section: Announcements. Nov. 24, 2020. URL: `https://aws.amazon.com/blogs/opensource/why-aws-loves-rust-and-how-wed-like-to-help/` (visited on 08/15/2023).

[13] *Welcome to Comprehensive Rust - Comprehensive Rust.* URL: `https://google.github.io/comprehensive-rust/` (visited on 08/15/2023).

[14] *Programming languages endorsed for server-side use at Meta.* Engineering at Meta. July 27, 2022. URL: `https://engineering.fb.com/2022/07/27/developer-tools/programming-languages-endorsed-for-server-side-use-at-meta/` (visited on 08/15/2023).

[15] jirehl. *Microsoft Azure CTO Wants to Replace C and C++ With Rust | The Software Report*. Oct. 21, 2022. URL: https://www.thesoftwarereport.com/microsoft-azure-cto-wants-to-replace-c-and-c-with-rust/ (visited on 08/15/2023).

[16] Thomas Claburn. *Linus Torvalds says Rust is coming to the Linux kernel*. URL: https://www.theregister.com/2022/06/23/linus_torvalds_rust_linux_kernel/ (visited on 08/15/2023).

[17] *bindgen*. original-date: 2016-06-22T15:05:51Z. Aug. 15, 2023. URL: https://github.com/rust-lang/rust-bindgen (visited on 08/15/2023).

[18] PyO3 Project and Contributors. *PyO3*. original-date: 2017-05-13T05:22:06Z. Aug. 15, 2023. URL: https://github.com/PyO3/pyo3 (visited on 08/15/2023).

[19] Tony Hoare. *Null References: The Billion Dollar Mistake*. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (visited on 08/28/2023).

[20] *Stack Overflow Developer Survey 2023*. Stack Overflow. URL: https://survey.stackoverflow.co/2023 (visited on 08/28/2023).

[21] *cargo bench - The Cargo Book*. URL: https://doc.rust-lang.org/cargo/commands/cargo-bench.html (visited on 08/28/2023).

[22] *When will benchmark testing be stable?* The Rust Programming Language Forum. Feb. 20, 2019. URL: https://users.rust-lang.org/t/when-will-benchmark-testing-be-stable/25482 (visited on 08/28/2023).

[23] Andrew Gallant. *cargo benchcmp*. original-date: 2016-02-15T01:05:58Z. Aug. 23, 2023. URL: https://github.com/BurntSushi/cargo-benchcmp (visited on 08/28/2023).

[24] *bheisler/criterion.rs: Statistics-driven benchmarking library for Rust*. URL: https://github.com/bheisler/criterion.rs (visited on 08/28/2023).

[25] *black_box in criterion - Rust*. URL: https://docs.rs/criterion/latest/criterion/fn.black_box.html (visited on 08/28/2023).

[26] Andrew Gallant. *critcmp*. original-date: 2018-09-18T21:42:02Z. Aug. 22, 2023. URL: https://github.com/BurntSushi/critcmp (visited on 08/28/2023).

[27] David Peter. *hyperfine*. Version 1.16.1. original-date: 2018-01-13T15:49:54Z. Mar. 2023. URL: https://github.com/sharkdp/hyperfine (visited on 08/29/2023).

[28] *Documentation for /proc/sys/vm/*. URL: https://www.kernel.org/doc/Documentation/sysctl/vm.txt (visited on 08/29/2023).

[29] David Peter. *hyperfine visualization scripts*. Version 1.16.1. original-date: 2018-01-13T15:49:54Z. Mar. 2023. URL: https://github.com/sharkdp/hyperfine/tree/master/scripts (visited on 08/29/2023).

[30] *Shnatsel/bounds-check-cookbook: Recipes for avoiding bounds checks in Rust, without unsafe!* URL: https://github.com/Shnatsel/bounds-check-cookbook/ (visited on 09/15/2023).

[31] Matt Godbolt. *Compiler Explorer*. Jan. 24, 2023. URL: https://godbolt.org/ (visited on 09/16/2023).

[32] pacak. *cargo-show-asm*. original-date: 2022-03-04T02:12:53Z. Sept. 15, 2023. URL: https://github.com/pacak/cargo-show-asm (visited on 09/16/2023).

[33] Egor Larionov. *Egor Larionov / unroll · GitLab*. June 6, 2022. URL: https://gitlab.com/elrnv/unroll (visited on 09/16/2023).

[34] *Code generation - The Rust Reference*. URL: https://doc.rust-lang.org/nightly/reference/attributes/codegen.html?#the-inline-attribute (visited on 09/16/2023).

[35] *Perf Wiki*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 09/16/2023).

[36] *Cachegrind: a high-precision tracing profiler*. URL: https://valgrind.org/docs/manual/cg-manual.html (visited on 09/16/2023).

[37] *Profiling - The Rust Performance Book*. URL: https://nnethercote.github.io/perf-book/profiling.html (visited on 09/16/2023).

[38] Ted Mielczarek. *luser/rustfilt*. original-date: 2016-05-13T17:00:31Z. Sept. 6, 2023. URL: https://github.com/luser/rustfilt (visited on 09/16/2023).

[39] *[cargo-]flamegraph*. original-date: 2019-03-07T16:31:30Z. Sept. 16, 2023. URL: https://github.com/flamegraph-rs/flamegraph (visited on 09/16/2023).

[40] Brook Heisler. *Iai*. original-date: 2021-01-02T20:54:31Z. Sept. 13, 2023. URL: https://github.com/bheisler/iai (visited on 09/16/2023).

[41] *Exploring PGO for the Rust compiler | Inside Rust Blog*. URL: https://blog.rust-lang.org/inside-rust/2020/11/11/exploring-pgo-for-the-rust-compiler.html (visited on 09/16/2023).

[42] *MPI bindings for Rust*. original-date: 2015-07-21T20:51:28Z. Sept. 16, 2023. URL: https://github.com/rsmpi/rsmpi (visited on 09/16/2023).

[43] Lars Quentin and Johann Carl Meyer. *walky*. Version 0.1.0. original-date: 2023-04-27T17:18:12Z. June 2023. URL: https://github.com/lquenti/walky (visited on 09/16/2023).

[44] Rust. *SIMD instructions with Rust on Android by Guillaume Endignoux - Rust Zürisee June 2023*. June 19, 2023. URL: https://www.youtube.com/watch?v=x5tK5ET6Q1I (visited on 09/16/2023).

[45] *Fearless Concurrency - The Rust Programming Language*. URL: https://doc.rust-lang.org/book/ch16-00-concurrency.html (visited on 09/16/2023).

[46] *Tokio*. original-date: 2016-09-09T22:31:36Z. Sept. 16, 2023. URL: https://github.com/tokio-rs/tokio (visited on 09/16/2023).

[47] *src/lib/fuchsia-async - fuchsia - Git at Google*. URL: https://fuchsia.googlesource.com/fuchsia/+/master/src/lib/fuchsia-async/ (visited on 09/16/2023).

[48] *Asynchronous Programming in Rust*. URL: https://rust-lang.github.io/async-book/ (visited on 09/16/2023).

[49] *RustLatam 2019 - Without Boats: Zero-Cost Async IO*. URL: https://www.youtube.com/watch?v=skos4B5x7qE (visited on 09/16/2023).

[50] Amos Wenger. *Understanding Rust futures by going way too deep*. fasterthanli.me. July 25, 2021. URL: https://fasterthanli.me/articles/understanding-rust-futures-by-going-way-too-deep (visited on 09/16/2023).

[51] Nicholas Nethercote. *The Rust Performance Book*. URL: https://nnethercote.github.io/perf-book/ (visited on 09/16/2023).

[52] *Algorithms for Modern Hardware - Algorithmica.* URL: `https://en.algorithmica.org/hpc/` (visited on 09/16/2023).

# A Measurement Data

## A.1 Static Sized

### A.1.1 Call By Value

| | **Lower bound** | **Estimate** | **Upper bound** |
|---|---|---|---|
| **Slope** | 307.55 ns | 310.15 ns | 313.04 ns |
| $R^2$ | 0.8472829 | 0.8537299 | 0.8458089 |
| **Mean** | 304.66 ns | 307.19 ns | 310.70 ns |
| **Std. Dev.** | 6.3231 ns | 15.613 ns | 24.335 ns |
| **Median** | 301.60 ns | 302.08 ns | 304.29 ns |
| **MAD** | 1.9063 ns | 3.0129 ns | 5.6761 ns |

Table 1: Full Data: Call By Value

### A.1.2 Call By Reference

| | **Lower bound** | **Estimate** | **Upper bound** |
|---|---|---|---|
| **Slope** | 204.67 ns | 205.86 ns | 207.17 ns |
| $R^2$ | 0.9323924 | 0.9360205 | 0.9315760 |
| **Mean** | 204.65 ns | 205.58 ns | 206.62 ns |
| **Std. Dev.** | 3.5931 ns | 5.0447 ns | 6.2856 ns |
| **Median** | 203.75 ns | 204.99 ns | 205.51 ns |
| **MAD** | 1.9471 ns | 2.8148 ns | 3.8151 ns |

Table 2: Full Data: Call By Reference

### A.1.3 Primitive Arrays

| | **Lower bound** | **Estimate** | **Upper bound** |
|---|---|---|---|
| **Slope** | 7.7883 ns | 7.8243 ns | 7.8659 ns |
| $R^2$ | 0.9481791 | 0.9509154 | 0.9472358 |
| **Mean** | 7.8054 ns | 7.8416 ns | 7.8845 ns |
| **Std. Dev.** | 125.05 ps | 202.72 ps | 279.05 ps |
| **Median** | 7.7467 ns | 7.7513 ns | 7.7607 ns |
| **MAD** | 16.400 ps | 24.632 ps | 44.433 ps |

Table 3: Full Data: Primitive Arrays

## A.2 Dynamically Sized

### A.2.1 No Compiler Optimizations, Naive

| | **Lower bound** | **Estimate** | **Upper bound** |
|---|---|---|---|
| $R^2$ | 0.0129440 | 0.0134225 | 0.0129313 |
| **Mean** | 23.550 s | 23.575 s | 23.600 s |
| **Std. Dev.** | 111.52 ms | 127.82 ms | 145.31 ms |
| **Median** | 23.494 s | 23.547 s | 23.636 s |
| **MAD** | 88.492 ms | 148.98 ms | 181.21 ms |

Table 4: Full Data: No Compiler Optimizations, Naive

### A.2.2 No Compiler Optimizations, Optimized code

| | **Lower bound** | **Estimate** | **Upper bound** |
|---|---|---|---|
| $R^2$ | 0.0131894 | 0.0136728 | 0.0131513 |
| **Mean** | 8.7616 s | 8.7684 s | 8.7756 s |
| **Std. Dev.** | 29.949 ms | 35.706 ms | 40.959 ms |
| **Median** | 8.7559 s | 8.7658 s | 8.7735 s |
| **MAD** | 24.535 ms | 30.151 ms | 40.061 ms |

Table 5: Full Data: No Compiler Optimizations, Optimized Code

### A.2.3 Compiler Optimizations, Optimized code

| | **Lower bound** | **Estimate** | **Upper bound** |
|---|---|---|---|
| $R^2$ | 0.1137963 | 0.1175455 | 0.1134823 |
| **Mean** | 3.0698 s | 3.0708 s | 3.0719 s |
| **Std. Dev.** | 4.2317 ms | 5.1644 ms | 6.0320 ms |
| **Median** | 3.0691 s | 3.0699 s | 3.0710 s |
| **MAD** | 3.3207 ms | 4.2208 ms | 5.5206 ms |

Table 6: Full Data: Compiler Optimizations, Optimized Code

### A.2.4 Compiler Optimizations, Optimized and Transposed code

| | **Lower bound** | **Estimate** | **Upper bound** |
|---|---|---|---|
| $R^2$ | 0.0054076 | 0.0056054 | 0.0053917 |
| **Mean** | 2.1711 s | 2.1714 s | 2.1716 s |
| **Std. Dev.** | 1.0805 ms | 1.3218 ms | 1.5562 ms |
| **Median** | 2.1709 s | 2.1713 s | 2.1717 s |
| **MAD** | 1.0773 ms | 1.4593 ms | 1.6688 ms |

Table 7: Full Data: Compiler Optimizations, Optimized and Transposed Code

# B Overview of all tools

| **Topic** | **Tool** |
|---|---|
| Microbenchmarking | Criterion |
| Application Benchmarking | Hyperfine |
| Assembly Generation | Compiler Explorer, cargo show-asm |
| Loop Unrolling | `unroll`, Compiler Arguments |
| Function Inlining | `#[inline]` |
| Statistical Profiling | `cargo-flamegraph` |
| CI benchmarking | `iai` |
| SIMD | `std::simd`, `core::arch` |
| Intra-Node parallelism | rayon |